(4)

Technical Document 1751
November 1989

Distributed Operating and Run-Time Systems: Mach and ARTX

Dale Brouhard
Manchi Gadbois
Raymond Liu

ADMINISTRATIVE INFORMATION

This work was performed by the Computer Systems Software and Technology Branch, Code 411, and the Distributed Systems Branch, Code 413, Naval Ocean Systems Center under a block program.

Released by
R. A. Wasilausky, Head
Computer Systems Software
and Technology Branch

Under authority of
A. G. Justice, Head
Information Processing
and Displaying Division

JG

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

1.0 DISTRIBUTED OPERATING AND RUN-TIME SYSTEMS

1.1 INTRODUCTION

The Next Generation Computer Resources (NGCR) Program is the Navy 6.3 program being proposed to provide computer resources to the Navy through the adoption of an open systems architecture. With the initiation of an open system architecture, it is expected that (more) rapid technology transition will occur. The Distributed Operating and Run-Time Systems task of the Mission Critical Computing Project is providing 6.2 research results on which to base the NGCR Operating Systems Standards Working Group (OSSWG).

The NGCR Program will provide the majority of computer resources to the Navy for military critical applications. Influencing the NGCR is an important payoff for Navy research. By providing a 6.2 research base, this task puts the NGCR OSSWG in a position to make technically knowledgeable decisions in a timely manner.

The objective of this task is to address research and technical issues (as identified in the NGCR OSSWG) in the area of distributed operating and run-time systems. Candidate systems will be evaluated with respect to their current level of maturity and to how well they address the NGCR's requirements for an operating system.

In FY89, the Mach Operating System (release 2.0) from Carnegie Mellon University (CMU) and Ready Systems' Ada Real-Time Executive (ARTX) were evaluated in the context of NGCR OSSWG issues. Mach is one of the more mature (advanced) projects that is addressing many of the relevant issues (distributed, extensibility, etc.) and it has been developed and promoted by Defense Advanced Research Projects Agency (DARPA). In the area of Ada Real-Time environments, Ready Systems' ARTX has greater maturity and availability than many other similar efforts. ARTX is also upwardly compatible with Ready Systems' industry standard real-time kernel, VRTX.

Section 2 describes the NGCR OSSWG abstract model used as a framework for evaluating the candidate systems. Sections 3 and 4 present Mach and ARTX in this framework. The results are summarized in section 5.

(KR) (--

2.0 NGCR OSSWG ABSTRACT MODEL

An operating system interface standard is one of the key elements in the success of NGCR. The function of the operating system is to control operation of all the computing system hardware and software elements in a coordinated, uniform manner consistent with the needs of embedded and real-time applications. The operating system capabilities include system initialization, fault tolerance and recovery, global resource allocation, and interprocess communication. The operating system interface standard is not a design of the operating system component of each processing system but is, in part, a specification of an application program interface common to all computing elements. (The appropriate specification level for this interface is yet to be determined.) The specification provides the basis for system-wide dynamic task and resource allocation. Global dynamic task and resource allocation is the basis for system-wide fault tolerance and recovery in heterogeneous processing systems. Some implementations of the Operating System Standards will provide the ability to achieve multilevel security at the system level.

The OSSWG Abstract Model is a conceptual model that provides a context for an application program developer's requirements for comparing existing operating systems and for standards specification. It provides a minimum, common set of conceptual embedded system building blocks with associated interfaces and functionality. Many of these system building blocks will be the results of other parts of the NGCR project.

The model is described from the application developer's perspective, i.e., the model records the embedded system developer's perception (mental model) of the overall large distributed, emɒedded system and its project support environment. This point of view is used so that

- application developers will have the proper services to meet their requirements and
- vendor implementation will not be constrained unnecessarily.

The following sections describe the major groups of operating system services in the Abstract Model that may be required of the NGCR Operating System. Not all of these services require a programming interface; therefore the services can be described as either explicit or implicit. Explicit services are those that can be accessed from an application program and, generally, are only provided when requested. Implicit services, on the other hand, are services that the operating system provides without a direct request. An example of an implicit service is the prevention of one program from writing over the memory of another. An example of an explicit service is a call to an operating system routine to output a block of memory to some device.

2.1 LANGUAGE SUPPORT SERVICES

Navy languages and their standard libraries have specific operating system needs that the NGCR Operating System Standard should meet. This section emphasizes the needs for Ada support because that language is currently required for all new weapon systems and major modifications to weapon system programs.

2.1.1 Ada Language Support Services

These seıvices support the use of the Ada programming language. While an NGCR Operating System Standard compliant operating system may be implemented in various languages, it should support the execution of programs written in Ada. At the least this means that the operating system together with the compiler's run-time library should include all necessary parts of an Ada Run-Time Environment (ARTE).

2.1.2 Support for Other Languages

Other languages have fewer requirements on the operating system than Ada. The C Language itself places almost no requirements on the operating system. The usual C Language libraries, however, require simple services like byte stream input/output (I/O) and the ability to create and receive signals. The CMS-2 language also has few requirements on the operating system. Lisp requires support for garbage collection, which can be provided by the hardware and the operating system or provided by the Lisp Run-Time system.

2.2 CAPABILITY AND SECURITY SERVICES

These services support the ability of the system to control usage so system integrity is protected from inadvertent or malicious misuse. These protection services provide a mechanism for the enforcement of the policies governing resource usage. Note that many of the security services are implicit services, i.e., they are provided without an explicit request to the operating system. There are two distinct classes of system access with which operating system services must be concerned: physical access and logical access.

2.3 DATABASE SERVICES

The database management system in an embedded system has several functions, including access control, consistency checks, maintaining consistent copies for fault-tolerance, and security. The need for Database Services as part of the Operating System Standard arises because of the interaction of the Database Management System's need for performance and multilevel security needs. If the operating system kernel is part of a Trusted Computing Base (TCB) for a multilevel secure system, the lower level parts of the database management system (the "database kernel") will have to be part of that TCB or be built "on top of" the operating system kernel.

2.4 DATA INTERCHANGE SERVICES

This set of services provides data conversion among different data representations. One scheme for providing these services is to have a single canonical representation for the important data types (integer, real, time etc.) and then each implementation of the operating system or compiler would provide conversion functions between the canonical representations and its own internal representations.

2.5 EVENT AND ERROR MANAGEMENT SERVICES

These services provide a common facility for the generation and communication of asynchronous events among the system and application programs. A major use of the event services is to report error conditions, but they may be used by device drivers and the operating system to provide an indication of some condition to the application programs.

2.6 FILE SERVICES

These services allow the system and applications to create permanent storage locations for data. The data are stored on files, and the files are organized in directories. Files are managed and accessed through logical names by the many system components that use the files, such as the application, system operator, and program support environment.

2.7 GENERALIZED I/O SERVICES

Generalized I/O services provide higher level constructs and functions for doing I/O to devices that do not fit well into the common file I/O paradigm. These services include nonblocking I/O and I/O to special devices. In nonblocking I/O, input or output is initiated under program control but the program continues execution while the transfer takes place. Many special hardware devices may need I/O supervised by the operating system.

2.8 MAN-MACHINE INTERFACE (MMI) SERVICES

These services allow I/O to be interchanged between the system and the user of the embedded system in an efficient and standardized way. Services that may be included are menu services, windowing services, command line services, parsing services, and pointer device services. These services will interface with low-level device services as needed, while presenting a higher level view to the human user. Higher level interfaces for much of this set of services may be provided by the Graphics Language/Interface Working Group (GL/IWG) of NGCR, rather than the OSSWG.

2.9 NETWORKS AND COMMUNICATIONS

These services involve the information exchange between the local processor nodes of an NGCR system. Network Control and Status services provide authorized users the capabilities to determine the status of network components and to control network working parameters.

Interprocess Communication services allow a local processor node's local operating system to request a procedure, function, or transaction to be performed on another processor node or logical resource. There are various forms of interprocess communication, some of which specify the receiver, some specify the sender, some are synchronous (i.e., delay the sender until the communication is completed), and some are asynchronous, etc. The particular forms that need to be specified by the NGCR Operating System Standard are yet to be determined.

2.10 PROCESS MANAGEMENT SERVICES

Typical process management services are required by application programs. For example: create a process and make it ready for execution; destroy a process and recover its resources; evaluate a reference to a process; and evaluate a connection to a process (where a connection is logical communication path between any two processes).

2.11 RELIABILITY, ADAPTABILITY, AND MAINTAINABILITY SERVICES

Robustness of a system or application is a desirable feature. The services supporting robustness (reliability, adaptability, and maintainability) are often implied services in that there is not a direct interface to these services through the application interface layer of the operating system. Reliability and adaptability services deal with the need for the system to perform functions that the application requests in a timely manner, whenever possible. Reliability is the ability to correctly perform a job to completion, adaptability is the ability to change the system's logical makeup (or jobs to do) over time, while maintainability is the ability to keep the system in operating condition. A highly adaptable system can facilitate the reliability of application's functions.

Software Safety services involve the system's ability to keep application software from causing harm to the system's software, hardware, or user. For instance, a process may attempt to write into another process's memory space without permission.

Status of System Components services involve the obtrusive and nonobtrusive diagnosis of the state of system components. These services may additionally need to record or display information concerning performance, configuration, and general system information.

Reconfiguration services allow the system to reconfigure its view of the world. These services allow the system to substitute different resources to perform system functions, such as substituting a new physical I/O channel to support a logical channel. These services are part of the programming interface but their use may be restricted to specially authorized programs such as those used by the system operator.

2.12 RESOURCE MANAGEMENT SERVICES

These services are involved in the management of the system resources. Resources include memory, I/O devices, and other physical devices. Services that manage the use of the central processing unit (CPU) are described in section 2.10, Process Management Services.

2.13 SCHEDULING SERVICES

These services schedule or arbitrate the use of various resources of the NGCR operating system, particularly the CPU. The scheduling services must be able to queue up requests to use a particular resource. This situation is made more complicated by the common need to schedule processes to run cyclically at a fixed period. When the resources become idle, the scheduler must select one of the "requestors" of the resources to have permission to use the resource. These services are listed separately, rather than under the services that use scheduling, to emphasize that there should be uniformity and consistency of scheduling across the range of resources.

2.14 SYNCHRONIZATION SERVICES

These services are involved in the ability to synchronize the operations of other services, functions, processes, and resources. Services such as distributed voting and remote resource allocation will need to use these services to accomplish their required functionality. Synchronization services are needed for both the local processor operating system's operation and the control of the distributed system.

2.15 SYSTEM INITIALIZATION AND REINITIALIZATION SERVICES

System initialization includes a complete restarting of the software, starting up the attached hardware subsystems devices, doing subsystem and system self tests, and completely clearing the database.

System reinitialization includes restarting the software while using the existing database information. The software may have to be reloaded, and the database may have been reestablished by a system recovery. Attached hardware subsystems devices may be reinitialized.

2.16 TIME SERVICES

The following time management services are some of those likely to be needed: Local Time of Day, which includes the time based upon a 24-hour or 12-hour clock; measurement of elapsed time; distributed time, which would be a capability to coordinate Local Time of Day maintained by local operating systems; and requests for process notification at a specific time or after a specified delay.

3.0 MACH

Mach is a UNIX-compatible operating system currently being developed at Carnegie-Mellon University and funded by DARPA as part of the Strategic Computing Initiative (SCI). The goal of the Mach effort is to create a simple, extensible kernel that supports an integrated, networked computing environment consisting of both large and small multiprocessors and uniprocessors.

The approach in Mach has been to design and build a kernel that is suitable for distributed systems and is also able to emulate UNIX. The UNIX emulation enables Mach to provide an environment in which users may continue to use programs with the UNIX system call interface. This is essential for practical reasons because so many UNIX application programs exist.

Mach is an open system based on lightweight kernel running in each computer with services such as the file system, network service, and process management outside the kernel. These services replace the system calls found in conventional operating systems such as UNIX. The model provided by Mach is a service model in which objects are managed by servers, and clients make request for operations on objects by using remote procedure calls. Remote procedure calling is supported by efficient and flexible interprocess communication facilities in the kernel.

Mach provides a binary compatible UNIX 4.3BSD interface, which includes a number of new facilities not available in 4.3:

- **Support for multiprocessors including**
  - provision for both tightly coupled and loosely coupled general-purpose multiprocessors ranging from (1) uniform access, shared memory multiprocessors (e.g., Encore Multimax, Sequent Balance), to (2) differential access, shared memory multiprocessors (e.g., BBN Butterfly, IBM RP3), to (3) multicomputer architectures (e.g., hypercube), and
  - separation of the process abstraction into an address space and resource abstraction, the *task*, and a processing abstraction, the *thread*, with the ability to execute multiple threads within a task simultaneously.
- **A new virtual memory design, which provides**
  - large, sparse virtual address spaces,
  - copy-on-write and read-write memory sharing between tasks,
  - memory mapped files, and
  - user-provided backing store objects and external pagers.
- **A capability-based interprocess communication facility**
  - transparently extendible across network boundaries with preservation of capability protection, and
  - integrated with the virtual memory system and capable of transferring large amounts of data up to the size of an address space via copy-on-write techniques.
- **A number of basic system support facilities, including**
  - an internal adb-like kernel debugger, and

Mach was built not as an extension of Berkeley UNIX facilities but as a new foundation upon which UNIX-like facilities can be built and future development of multiprocessor systems for new architectures can continue. The current, evolved vision is for the Mach distributed operating system to be based on a minimal kernel upon which multiple operating system environments can be built. In a minimal kernel, a large part of the traditional operating system code is implemented in user processes, with the kernel handling the communication between clients and servers. At this point, the kernelization is not complete, and some UNIX functionality is still embedded in Mach kernel code. When the kernelization is complete, it will be possible to emulate operating system environments other than UNIX 4.3 BSD on top of the Mach kernel.

3.1 LANGUAGE SUPPORT SERVICES

The need ıɔr Ada support and other languages support is described in this section.

3.1.1 Ada Language Support Services

Several Mach mechanisms provide support for Ada (an objective of the Mach project is to support Ada). Mach threads (lightweight processes) are a good match for handling Ada tasking. Mach's flexible virtual memory management and message passing mechanisms (provide interprocess communication, exception handling) form a basis for an Ada run-time system. The more direct support the underlying computing resource provides the generated Ada program, the smaller the needed run-time system support for that system.

3.1.2 Support for Other Languages

In addition to 4.3BSD UNIX compatibility, Mach provides C and C++ kernel interface libraries.

Mach also provides Matchmaker, which is a language for specifying and automating the generation of multilingual interprocess communication interfaces. The Mach Interface Generator (MIG) is an interim implementation of a subset of the Matchmaker language that generates remote procedure call interfaces for interprocess communication between Mach tasks. The Mach environment currently supports the languages Common Lisp, C, C++, Ada, and Pascal.

The user provides a specification file defining parameters of both the message passing interface and the procedure call interface. MIG then generates three files:

- **User Interface Module**, which is compiled with the client program.
- **User Header Module**, which is to be included in the client code to define the types and routines needed a' compilation time.
- **Server Interface Module**, which is compiled with the server process.

3.2 CAPABILITY AND SECURITY SERVICES

This section presents the services that support the ability of the system to control use so system integrity is protected from inadvertent or malicious misuse.

3.2.1 Naming and Protection

Each task makes use of a collection of resources such as areas of memory, files, and ports (communication channels). The resources belonging to a task need some form of protection against the actions of other tasks in the same computer. One approach for protecting files and ports is the use of capabilities. A capability is a reference or name that acts as a token providing access to a resource or data object. When a task possesses the capability for a resource or data object, it is allowed access to it.

The Mach kernel uses capabilities for naming and protection on a single system. A capability is an identifier for a stored data object or a system resource that also grants rights to perform certain operations on the object or resource. The rights that a capability grants must be either protected or hidden so that user processes cannot alter them to grant themselves rights. Ports are used to refer to objects, and operations on objects are requested by sending messages to the ports which represent them.

Access rights to a port consist of the ability to **send to, receive from,** or **own** that port. A task may hold just send rights or any combination of receive and ownership rights plus send rights. Threads within a task may only refer to ports to which that task has been given access. When a new port is created within a task, that task is given all three access rights to that port.

Port access rights can be passed in messages. They are interpreted by the kernel and transferred from the sender to the kernel upon message transmission and to the receiver upon message reception. The network message servers extend the protection to the network environment by implementing mechanisms to protect both the messages sent over the network to network ports and the network port capabilities.

The network servers maintain a space of *Network Ports* and each network server maintains a mapping between ports local to its host and network ports. Each network port is represented by a *Network Port Identifier,* which contains information to locate the receiver and owner for the network port and information, allowing the security of the Mach port abstraction to be maintained in the network environment.

Each network server holds receive rights to those network ports for which the receive rights to the corresponding local ports are held by local tasks. Send and ownership rights to network ports are handled similarly, except that send rights to a network port may be held by many network servers.

3.2.2 Security-Trusted Mach

The Trusted Mach project is a DARPA-sponsored research effort of Trusted Information Systems, Inc.. The goal is to build a version of Mach-Trusted Mach - that meets the B3 level of protection as specified in the National Computer Security Center (NCSC) Trusted Computer System Evaluation Criteria (TCSEC), the so-called "Orange Book."

Trusted Information Systems ascertained that Mach's design made implementation of classification labels and access control lists easier than in traditional UNIX. The design separation of the kernel and services make modification of the operating system more straightforward and easier to verify as being a trusted system.

The project adopts the idea of "incremental reference monitors." At the lowest level is the Trusted Mach Kernel. At the intermediate level is the reference monitor composed of the kernel and a trusted name server. At the highest level is the reference monitor composed of the kernel, a trusted name server, and other trusted servers. Thus far, work has concentrated on the kernel level of a single machine. Mach's ports are serving as the protected objects in Trusted Mach; its tasks (through their threads, which are the active entities) are serving as the subjects. Extensions are being developed to meet the TCSEC requirements for both discretionary and mandatory protection.

At this time, the Trusted Mach project is using a Spring 1988 version of Mach. Since this version is not kernelized, the effort cannot yield a trusted operating system. The unkernelized version of Mach is serving as a platform for research into multilevel security, not as a base upon which to build a trusted system. The development of a trusted version is tied to the completion of Mach kernelization.

3.3 DATABASE SERVICES

Distributed transactions are an important technique for simplifying the construction of reliable and available distributed applications. Transactions provide failure atomicity, permanence, and serializability, all of which reduce the amount of attention an application programmer must pay to concurrency and failures.

Camelot, built on top of Mach, supports the execution of distributed transactions and the definition, management, and use of data servers, which encapsulate shared, recoverable objects. Camelot is based on the client-server model and uses remote procedure calls both locally and remotely to provide communication among applications and servers.

Mach provides the most basic blocks for reliable distributed applications - communication facilities and the MIG stub generator. Camelot provides functions that support both blocking and nonblocking commit protocols, nested transactions, and a scheme for supporting recoverable objects that are accessed in virtual memory:

- System Configuration Management - Camelot maintains configuration data so it can restart the appropriate data servers after a crash and reattach them to their recoverable storage. These configuration data are stored in recoverable storage and updated transactionally.
- Disk Management - With the cooperation of Mach, Camelot permits data servers to map recoverable storage into their address space. Data servers are responsible for doing their own microscopic storage management. Camelot provides data servers with logging services for recording modifications to objects so operations on data in recoverable storage can be undone and redone after failures.
- Recovery Management - Camelot's recovery functions include transaction abort, server, node, and media-failure recovery.
- Transaction Management - Camelot provides facilities for initiating, committing, and aborting top-level and nested transactions: *Blocking* and *Nonblocking*. Camelot also provides an inquiry facility for determining the status of a transaction. This is used to support locking in nested transactions.
- Reliability and Performance Evaluation - Camelot will contain a facility for capturing performance data, generating and distributing workloads, and inserting (simulated) faults.

3.4 DATA INTERCHANGE SERVICES

The Matchmaker interface language and compiler is used in the Mach environment. It was designed to allow distributed programs to be built from multiple existing programming languages by using remote procedure calls (RPC). The Mach environment currently supports the languages Common Lisp, C, C++, Ada, and Pascal. Both clients and servers can be built in any of these languages.

Matchmaker is built on top of objects represented by ports and operations invoked by RPC message, but Matchmaker hides these from the programmer, allowing services to be defined by procedural interfaces. The interface language allows clients and servers in several different languages to communicate. Specifically, it provides

- support for multiple, existing programming languages,
- language support for object references,
- language interfaces for object operations,
- language and machine independent operation interface specifications, and
- automated interface code generation from interface specifications.

The Matchmaker language is used to mask language differences by compiling object-oriented interprocess communication interface specifications into client and server RPC code implementing those interfaces for each target language. Matchmaker handles differences in language syntax, type representations, record field layout, procedure call semantics, and exception handling semantics.

Differences in type representation by various programming languages within each machine are handled by Matchmaker. Data representation issues across machine boundaries are handled through message server processes. Byte reordering and machine specific conversions are performed by the message servers with the responsibility for conversion always resting with the receiving host.

3.5 EVENT AND ERROR MANAGEMENT SERVICES

External events that affect the execution of a program are called exceptions. Mach has taken a generalized view of all exceptions. An exception requires suspension of the "victim" thread that caused the event and notification of an exception handler. The handler performs some operations as a result of the exception, and then the victim is either revived or terminated. Because the handler is never within the victim thread, all the exception handling involves a form of remote procedure calls. Mach ports and messages are the elements through which all this happens. The handler's port for communicating with the task is the thread (or task) exception port.

When a thread raises an exception, a message is sent to its thread exception port to notify its error handler, which executes in a separate thread. If no handler exists or the handler fails to recover the exception, the message is forwarded to the exception port of the task in which the exception-incurring thread exists.

A debugger can intercept unhandled exceptions for all threads in a task by attaching itself to the task exception port. This enables a debugger to coexist with error handlers, in that the debugger is aware only of those exceptions not handled by an error handler. The Mach kernel has a built-in kernel debugger (kdb) based on adb (a UNIX debugger).

This design provides a single facility with a consistent method of handling all exceptions, a simple interface, full support for debuggers and error handlers, and no duplication of functionality within the kernel. In addition, this design allows for user-defined exceptions.

The network server keeps a log in memory of various events happening during its operation. This log, along with statistics on various operations, can be obtained via the *logstat* service exported by the network server. In addition, many operating parameters, including the level of debugging information written to the log, can be set using this same service.

3.6 FILE SERVICES

The current vision is for Mach to be based on a minimal kernel upon which multiple file systems can be built. At this point, the kernelization is not complete, and the UNIX file system (4.3BSD) functionality is still embedded in Mach kernel code. The next Mach release (2.5) will include Sun's Network File System (NFS) and the Andrew File System.

Mach provides memory-mapped files, which are files that can be mapped into the virtual address space of a task. Users of memory-mapped files can treat file data in the same way as normal memory. This provides a single-level storage system available to tasks outside the kernel. In the Mach UNIX implementation, program loading is achieved by memory-mapped files. Mach also provides conventional files, and the standard I/O library has been implemented twice (using memory-mapped files and using normal files) with a useful performance gain for memory-mapped files. In the standard I/O library, when a file is opened, the whole file is mapped into the caller's address space. Although the semantics of the library remain the same, the data in write operations are saved in the main memory and are not written to the file until the file is flushed or closed.

When virtual memory is allocated, an external pager may be chosen, and for a memory-mapped file the choice is a file system. The request from a kernel to an external pager to provide pages for paging in and to receive pages that are paged out are remote procedure calls and are addressed to a port. Therefore, the external pager may be used by any number of kernels that are either local or remote. This enables memory-mapped files to be shared by tasks running in computers distributed throughout a network. Mach provides functions to enable a kernel and an external pager to cooperate in the use of shared files. The data manager can request the kernel to accept new copies of pages or to give it the latest versions of pages and to lock pages.

3.7 GENERALIZED I/O SERVICES

Currently, the UNIX I/O libraries are used with low-level device drivers residing in the Mach kernel. As mentioned above, the standard I/O library has been implemented twice (using memory-mapped files and using normal files) with a useful performance gain for memory-mapped files.

3.8 MAN-MACHINE INTERFACE (MMI) SERVICES

X-Windows is used as a basis for the MMI along with the UNIX shells.

3.9 NETWORKS AND COMMUNICATIONS

Mach provides a flexible interprocess communication facility. To facilitate building distributed systems, Mach stresses

- a capability based interprocess communication paradigm,
- messages containing typed data,
- transparent extension of local communication into a network by network message servers,
- an interface language, Matchmaker, which is used to generate client/server interfaces, and
- integration with virtual memory management for efficient transfer of large messages.

Mach interprocess communication facility, which is defined in terms of *ports* and *messages*, provides location independence, security, and data-type tagging.

3.9.1 Message Passing

A *message* consists of a fixed-length header and a variable-size collection of typed-data objects. Messages may contain both port capabilities or embedded pointers so long as both are properly typed. A single message may transfer up to the entire address space of a task. Message-passing is the primary means of communication both among tasks and between tasks and the operating system kernel itself. Messages may be sent and received either synchronously or asynchronously. A task sends a message by using the *Send* operation and another task receives it by using the *Receive* operation on the same port.

Both blocking and nonblocking communication may be used. In the case of blocking communication, the message is transferred directly to the recipient task at the time when it invokes the *Receive* operation. For nonblocking communication, the message is passed in two stages, by copying from sender to kernel and from kernel to recipient. In both blocking and nonblocking communication, large messages (a few megabytes) may be transferred efficiently using the copy-on-write mechanism. In nonblocking communication, the message will reside temporarily in the kernel address space without being written and then will eventually be transferred to the recipient's address space.

A third primitive *rpc_message* is designed for making RPCs. It allows the invoking thread to send a call message and then to receive a reply. It includes a specification for timeouts on sending and receiving. The use of a special RPC primitive at the message level in this way simplifies the RPC protocol. The request message contains a *RequestId* enabling the transport layer in the server to return the replay to the caller.

Finally, the interprocess communication mechanism makes use of the virtual memory system to perform virtual copies of larges message rather than physical copies. This allows large amounts of data to be sent copy-on-write. This is an especially important feature since data received by a task are usually only read, requiring no copy.

3.9.2 Ports

A port is a communications channel. Application programs communicate with objects managed by the kernel and server tasks through the objects' ports. This is the software counterpart to the communications ports on the hardware. An object is said to have "access rights" to a port if it has dealings with that port.

The *port* is the basic transport abstraction incorporated by Mach. As a protected kernel object into which messages may be placed by tasks and from which messages may be removed, a port can be seen as a finite-length queue of communications sent by a task. Ports may have any number of senders but only one receiver. Access to a port is granted by receiving a message containing a port capability (to either **send** or **receive**).

Every task has its own port name space, used for port and port set names. Typically these names are small integers, but are implementation dependent. When a task receives a message carrying rights for a new port, the Mach kernel is free to choose any unused name, but a task's name for a port can be changed by a procedure call.

Tasks may create ports for their own use and, initially, all rights of access belong to the creator. To establish communication, tasks may send port capabilities in message to other tasks. A cluster can use the send and receive operations on a port only if it holds a capability for it. Capabilities grant one of the following access rights: *Receive, Owner,* and *Send*. A task can only get access to a port created by another task by receiving a capability in a message sent by another task. Any of the above rights may be conferred by the capability.

Both tasks and threads have a special kernel port by which the kernel recognizes them. Some special types of ports are associated only with tasks: the notify port, through which the task receives messages from the kernel about its port access rights and the status of messages it has sent; the exception port, through which the task receives messages from the kernel when an exception occurs; and the bootstrap port, with which new tasks attach to any services that they need.

Threads also have some special types of ports: the thread reply port, for early messages from a young thread's parent and early remote procedure calls; and the thread exception port, similar to the task exception port. Ports can be strung together into port sets, through which several objects can grab any messages from a single message queue.

3.9.3 Network Communication

The Mach facilities for communication over a network are provided by network servers outside the kernel that support message passing using port capabilities, so that communicating tasks cannot tell whether they are in the same computer or linked by a network. The Mach kernel itself has no knowledge of networks. As far as the kernel is concerned, messages are always passed between tasks on the same host. A network server acts as a local representative for tasks on remote sites, transparently extending communication over a network. Messages destined for ports with receivers in remote computers are sent to the local network server and clients cannot tell whether a port is local or remote.

Network servers communicate with one another via network ports. A network port is accessible to network servers in several computers and is addressed by a *network port identifier*. Network servers store mappings from ports used by tasks on the local computer and the corresponding network port identifiers. To send a message from a client to a server, the following steps take place:

- The client sends a message to a local port (known to the server port) and it is received by the network server.
- The network server uses its mappings to translate the local port to the network port identifier of the network server on the server computer. It then sends the message to that network port.
- The destination network server uses its mapping to obtain the local port of the server and sends on the message.

In addition to simply extending the interprocess communication paradigm to the network, network servers may participate in data type conversion and provide secure network transmission. By providing this functionality outside of the kernel, Mach allows a host more flexibility in choosing data type representations, the amount or type of security to be used in a network, and even the protocols to use for network transmission.

There are two standard servers that support use of Mach-style communications. One is the **Netmsgserver.** It passes all the Mach interprocess communication messages between machines. It also provides network wide port register and lookup functions.

The other general-purpose Mach server is the **Environment Manager.** It can register or look up ports or named strings but does not communicate with other Environment Managers.

In general, one decides to register a port with the Netmsgserver if it is to be known by tasks on arbitrary remote machines within the local network. Ports are registered with the Environment Manager if they are to be used only by tasks that share access to the same Environment Manager. Often such tasks are part of the same creation tree or are performing a computation on a single node.

3.9.4 Remote Procedure Call

The procedure call is a well-understood mechanism for communication within a program. In the design of conventional programs, procedures play an important role in decomposing a program into separate components. The remote procedure call (RPC) is modeled on the local procedure call, but the called procedure is executed in a different task and usually a different computer from the caller. The task that calls a remote procedure is referred to as a client and the task that executes the procedure as a server.

The RPC can be used to build distributed programs in a way that is similar to the use of local procedure calling in conventional programs. An RPC is achieved by an exchange of request and reply messages between client and server containing the name of the procedure and its input or output arguments. Lightweight message passing protocols can enhance the performance of RPCs. The client that makes an RPC uses a binder to locate a server that has previously registered the service with that binder.

Interface specifications are required in RPC systems, and the Matchmaker interface language and compiler is used in the Mach environment. It was designed to allow distributed programs to be built from multiple existing programming languages by using RPCs. Matchmaker is built on top of objects represented by ports and operations invoked by RPC message, but Matchmaker hides these from the programmer, allowing services to be defined by procedural interfaces. The interface language allows clients and servers in several different languages to communicate.

RPCs are more vulnerable to failure than local calls, since they involve a network, another computer, and another task. They consume more time than local ones (100 to 1000 times greater). It can be argued that programs that make use of RPCs must handle errors that cannot occur in local procedure calls; hence, that RPC syntax should not be transparent and the language should be extended to make remote operations explicit to the programmer.

RPC has been implemented in UNIX systems, but performance is limited both by the large amount of copying required and by the characteristics of UNIX processes. Considerably better performance can be achieved in systems based on lightweight processes with shared memory and specialized message passing protocols.

In UNIX, the number of copies per request or reply message includes copying from the client program to the stub, from the stub to the kernel, and from the kernel to the network buffer with a similar number of copies on arrival. The Mach kernel was designed to achieve a high rate of data transfer by reducing the number of copying operations and by the use of message passing primitives designed to support remote procedure interfaces.

3.10 PROCESS MANAGEMENT SERVICES

Mach divides the process abstraction into two orthogonal abstractions: the task and the thread. A task is a collection of system resources, including a virtual address space and a set of port rights. A task provides a framework in which a number of threads carry out computations with the task as the unit of protection. The threads within a task are not protected from one another.

Tasks may be created, terminated, suspended, and resumed by other tasks. Task creation in this manner results in sets of tasks related in tree structures. When a new task is created, it is given capabilities for ports by its parent task. A task may inherit regions of virtual memory from its parent for use as read-write or copy-on-write.

A thread is an entity (an object) capable of performing computation, and for low overhead, it contains only the minimal state necessary. Another term for a thread is a lightweight process. All threads within a task share the virtual memory address space and communications privileges associated with their task.

The thread is the basic unit of computation; it is the specification of an execution state within a task. Mach allows multiple threads to execute within a single task. A thread can execute independently, and the threads within a task share the memory and ports belonging to that task. The thread is the unit of scheduling, and threads may be created, suspended, resumed, or terminated. The scheduling of threads within a task is subservient to that of the whole task. In a parallel processor, multiple threads from one task may run at the same time as one another, within the address space of the task. Operations on tasks and threads are invoked by sending a message to a port representing the task or thread.

The use of threads with shared memory is a desirable component of a distributed operating system kernel. The sharing substantially reduces the time required to create a process or to change the process that is currently active (context switch).

Application tasks can be programmed using concurrent threads. There are two main advantages over a UNIX program: (1) if a blocking call (e.g., for input/output) occurs, one thread can wait while another thread continues to work, whereas in UNIX the entire process is blocked, and (2) there is only one method of communication between tasks/threads-by sending messages, whereas, in UNIX, kernel threads communicate via shared memory and user processes must use pipes.

The use of threads and the design of system kernels to support them are perhaps the most significant factors in the success of most of the distributed systems that have been developed to date.

3.11 RELIABILITY, ADAPTABILITY, AND MAINTAINABILITY SERVICES

Distributed transactions are an important technique for simplifying the construction of reliable and available distributed applications. Transactions provide failure atomicity, permanence, and serializability, all of which reduce the amount of attention an application programmer must pay to concurrency and failures.

- **Failure atomicity** ensures that if a transaction's work is interrupted by a failure, any partially completed results will be undone. A programmer can then attempt the work again in entirety by reissuing the same or a similar transaction. This property simplifies the implementation of most replication algorithms and makes it easier to achieve high availability.

- **Permanence guarantees** that if a transaction completes successfully, the results of its operations will never be lost, except in the event of a catastrophe.

- **Serializability assures** that while transactions are allowed to execute concurrently, the results will be the same as if the transactions are executed serially. Other concurrently executing transactions cannot observe inconsistencies. Programmers are, therefore, free to cause temporary inconsistencies during the execution of a transaction knowing that their partial modifications will never be visible.

Mach provides the most basic blocks for reliable distributed applications - communication facilities and the Matchmaker interface. Camelot provides functions that support both blocking and nonblocking commit protocols, nested transactions, and a scheme for supporting recoverable objects that are accessed in virtual memory.

3.11.1 Camelot and Avalon

Camelot is a distributed transaction processing facility built on top of Mach. As such, it addresses the requirements of reliability and fault-tolerance. Its basic abstraction is the transaction. See section 4.3, Database Services.

Avalon, a set of language facilities built on top of Camelot and Mach, provides linguistic support for reliable applications based on atomic transaction. It is implemented as a preprocessor for C++, Common-Lisp, and Ada and automatically generates necessary calls on Camelot.

3.12 RESOURCE MANAGEMENT SERVICES

3.12.1 Memory Object

Each task's address space is used for the memory objects and also for messages and memory-mapped files. When a task allocates regions of virtual memory, the regions must be aligned on page boundaries. The task can create memory objects for use by its threads; these can actually be mapped onto the space of another task. Spawning new tasks is more efficient because memory does not need to be copied to the child. The child needs only to touch the necessary portions of its parent's address space.

3.12.2 Virtual Memory

The Mach virtual memory system provides the programmer with a clean interface, which allows virtual memory to be allocated and deallocated at arbitrary addresses and sizes, restricted only by the page size of the underlying hardware. Applications can, on a page-by-page basis, specify access modes such as read-only, read/write, or shared. Also on a page-by-page basis, virtual memory can be shared between tasks in a controlled fashion based on inheritance.

Regarding memory/communication integration, the Mach project emphasizes the complementary roles that memory and communication can play. Namely, Mach uses memory mapping techniques (i.e., copy-on-write sharing) to accomplish communication; an entire address space may be sent in a single message with no actual data copy operations performed. In the other direction, Mach implements virtual memory through its interprocess communication (IPC) facilities; in particular, it maps process addresses onto memory objects, which are represented by ports and accessed via messages. This is what enables user-provided memory objects.

When a new task is created, the parent task may specify that the new child task can share (read-write) or copy any portion of its address space. This enables separate tasks in the same inheritance tree (same computer) to make use of shared memory. Copy-on-write sharing is used to make virtual memory copying and message passing efficient. A copy-on-write page is shared between two clusters until one of them writes to it. At that time, a new page is placed in the address space of the writing task in place of the shared page.

Mach allows tasks outside the kernel to handle virtual memory management functions for "paging in" (providing pages of memory objects to be copied into main memory) and "paging out" (taking charge of pages that are removed from main memory). When virtual memory is allocated, an *external pager* may be specified to handle paging requests - if no external pager is specified, a default one is used. An external pager is a task that has access to a memory object generally stored on a disk. For a memory-mapped file, a file system is chosen for the pager. When a new page is required, the kernel requests data from the file system, and when an altered page is to be "paged out," the kernel gives it to the file system. The use of a file system for a default pager removes the need for separate paging and filing.

External memory management allows Mach to be extended in powerful ways without changing the base Mach kernel. For example, network consistent shared memory can be implemented by an external memory manager. The shared memory manager can use the external memory interface to control which pages of memory can be accessed by which machines at various times to guarantee control. Not only does this remove that complexity from the kernel, but it allows the shared memory manager to choose which algorithms it uses to enforce consistency and security.

3.13 SCHEDULING SERVICES

The Mach kernel implements a time-sharing scheduling algorithm similar to that of UNIX, which had to be modified for multiprocessor architectures. Also, an interface for external management of multi-processor scheduling is being considered.

The Advanced Real-Time Technology (ART) and Mach groups at CMU are working together on an experiment called RT-Mach (Real-Time Mach) in an attempt to transition some of the ART theoretical work into Mach. They are implementing ART's real-time thread model and scheduler on Mach, along with tools to predict the run-time behavior and worst-case blocking time during actual real-time system design. This will be for the kernelized version, which is still in development. RT-Mach is being targeted for medium real-time (scheduled events timing 100 ms or greater), so it maybe inappropriate for "fast" real-time (robot control or avionics).

3.14 SYNCHRONIZATION SERVICES

Mach provides constructs for protection of critical regions and synchronization. Lock and unlock primitives are used with mutex variable to provide mutual exclusion. Wait and signal primitives are used with condition variables to provide synchronization.

At a higher level, MIG provides a synchronous/asynchronous RPC interface.

When building a shared memory multiprocessor, care is usually taken to guarantee automatic cache consistency or at least to provide mechanisms for controlling cache consistency. However, hardware manufacturers do not typically treat the translation look-aside buffer (TLB) of a memory management unit as another type of cache that also must be kept consistent. To guarantee such consistency when changing virtual mappings, the kernel must determine which processors have an old mapping in a TLB and cause it to be flushed. However, it is impossible to reference or modify a TLB on a remote CPU on any of the multiprocessors that run Mach. Some possible solutions are employed by Mach in different settings:

- *Forcibly interrupt all CPUs that may be using a shared portion of an address map so their address translation buffers may be flushed* — this applies whenever a change is time critical and must be propagated at all costs.

- *Postpone use of a changed mapping until all CPUs have taken a timer interrupt (and had a chance to flush)* - this can be used by the paging system when the system needs to remove mappings from the hardware address maps in preparation for pageout.

- *Allow temporary inconsistency* - this is acceptable because the semantics of the operation being performed do not require or even allow simultaneity. For example, it is acceptable for a page to have its protection changed first for one task and then for another.

3.15 SYSTEM INITIALIZATION AND REINITIALIZATION SERVICES

The network server initialization sequence takes care of detecting modules that require kernel support not present on the current node and of setting the working parameters accordingly. These include

- Access to a network interface. If there is no network, the network server degenerates into a single local Name Server.
- Netport support with which kernel port records may be flagged as corresponding to local representatives for remote network ports.
- Versatile Message Transport Protocol (VMTP) support. The VMTP module creates a single well-known server entity to receive all network requests and keeps a pool of client entities to use on the client side to initiate transactions.

3.16 TIME SERVICES

Time services such as local-time-of-day service is based on UNIX.

The control structure of the network server is structured as a collection of threads sharing the same address space. Each thread is used to perform one specific task asynchronously with the other threads. There is a **Timer** thread, which is used by all the other threads whenever they need to schedule some action to take place at some given time in the future.

The code structure of the network server is distributed between several modules, each pertaining to some specific set of related operations or management of some data structure. One of the modules is the **Timer Service**, which accepts requests from other modules for events to be scheduled at some time in the future. When the event's deadline expires, the timer module calls the user-supplied function associated with the timer.

4.0 ARTX

Ready Systems' Ada Real-Time Executive (ARTX) is designed to implement the critical "kernel" services of an Ada multitasking real-time Run-Time System for embedded microprocessor applications (Motorola 68000 microprocessors family and Intel 80386 microprocessor). ARTX schedules the processor and allocates CPU time among a number of concurrent tasks; it also allocates blocks of available memory to these tasks and implements intertask communication and synchronization. Furthermore, it supports a full range of Ada semantic operations, including the complete Ada tasking model. Its real-time capabilities may be elaborated as follows:

- ARTX consists of deterministic algorithms with fixed, specified timing for task rescheduling, rendezvous calls and accepts, memory allocation, interrupt latency, and interrupts-off time.
- ARTX's timing is independent of system load. That means that, as system requirements change and more capabilities (tasks or other system objects) are added to the system, ARTX's timing is not affected.
- ARTX supports a fully preemptive scheduler so the highest priority task in the system will always be executing.
- ARTX allows task priority to be changed at run-time.
- ARTX provides additional communication and synchronization primitives besides the standard Ada rendezvous. These primitives are accessed from the applications code via a packaged interface and include mailboxes, queues, semaphores, and event flags.
- ARTX supports two alternatives for servicing interrupts: using Ada flexible rendezvous entries or using Ada procedure calls, which are fast but not as flexible.

ARTX is also upwardly compatible with Ready Systems' industry standard kernel VRTX32, so application tasks written in other languages (C, Fortran, and assembly language) can be integrated easily into the system without changes.

4.1 LANGUAGE SUPPORT SERVICES

4.1.1 Ada Language Support Services

ARTX supports complete Ada tasking operations (e.g., rendezvous, task creation and termination, task completion and abort, select, and delay). It also supports full-range Ada semantic operations (e.g., exception handling, input/output, and dynamic memory allocation).

4.1.2 Support for Other Languages

ARTX supports mixed-mode applications where a number of tasks may coexist in the same target system, may be written in a variety of languages (for example C, Fortran, Pascal, and assembly language), and may interact in various ways through the underlying ARTX capabilities (because ARTX is upwardly compatible with VRTX32).

4.2 CAPABILITY AND SECURITY SERVICES

ARTX does not explicitly address these services beyond the standard Ada semantics.

4.3 DATABASE SERVICES

ARTX does not explicitly address these services beyond the standard Ada semantics.

4.4 DATA INTERCHANGE SERVICES

Differences in type representation by various programming languages (Ada and C) within the ARTX environment are handled by the cross Ada and C compilers.

4.5 EVENT AND ERROR MANAGEMENT SERVICES

ARTX uses Event Flag to signal occurrences of events to tasks. It also detects event overruns, and it provides synchronization features such as:

- a task can wait for one of several events to occur,
- a task can wait for a total of several events to occur, and
- many tasks can be waiting for the same events to occur.

ARTX also uses the Ada exception mechanism for communicating error conditions. Exception handling features a complete traceback of the exception propagation. It provides information on the entire call chain of the exception, extending all the way to the originating Ada source line.

4.6 FILE SERVICES

4.6.1 Naming and Directory Services

ARTX can be extended to provide file services by including the Ready Systems' Input/Output File Executive (IFX). IFX is a file manager to provide file and directory handling functions for disk devices. The Disk I/O module of the IFX provides the MS-DOS compatible File Manager. The disk file operations are as follows:

- formatting and initialization,
- file management calls (open, close, create, delete, and rename files),
- operations on directories (make and remove),
- file locking,
- volume mounting and dismounting, and
- I/O operations (transferring data to and from files).

4.6.2 Real-Time Files

ARTX's IFX provides a buffer cache manager which reduces I/O operations to disks by maintaining a cache of frequently used sectors. Also, Interrupt Service Routines can make direct calls to IFX resulting in better serial operations.

4.7 GENERALIZED I/O SERVICES

IFX's Stream I/O module handles I/O for byte-stream devices such as terminals, printers, pipes, and other serial communications devices. The Stream I/O module consists of the Circular Buffer Manager and the Line Editor. The Buffer Manager is used for the high-speed binary communication between computers. The Line Editor is used with CRT terminals and printers.

4.8 MAN-MACHINE INTERFACE SERVICES

ARTX can be extended to provide a system monitor facility by including the Ready Systems' ARTX System Monitor and Real-Time Multitasking Debugger (RTscope). RTscope supports interactive ARTX system calls and displays ARTX system status. It also sets, displays, and removes breakpoints with qualifiers on any ARTX system call.

4.9 NETWORKS AND COMMUNICATIONS

ARTX can be configured to provide a multiprocessor networked run-time environment by using its two companion components RTAda-MP and RTAda-Net. RTAda-MP is for shared memory multiprocessing, and RTAda-Net is for multiprocessor communication over local area networks.

RTAda-MP supports a multiprocessor system organized as a "cluster." A cluster consists of two or more processors with local memory global or shared memory; a system bus that connects the processors; ARTX and RTAda-MP on each processor; and Ada application software. It provides three layers of interprocess communication and synchronization services. The lowest layer is the Physical Layer, which provides unbuffered communication between nodes. The Channel Layer is built on top of the Physical Layer and provides node-to-node message passing mechanism (channel is a buffered, virtual connection between two Ada tasks on two different processors that allows the tasks to send and receive messages). The highest layer is the RPC, which allows interprocessor procedure calls synchronously or asynchronously.

RTAda-Net allows the ARTX-based system to network with other systems that support terminal central processor/input processor (TCP/IP) and sockets (i.e., Sun/UNIX). For example, real-time applications in an RTAda-Net environment can use ARTX to gather, control, and allocate real-time processes while a more familiar interface (such as VAX/VMS with TCP/IP, or UNIX BSD) can be used for data storage, or as a system monitor in a non-real-time environment.

4.10 PROCESS MANAGEMENT SERVICES

ARTX supports the complete Ada tasking model. It provides system calls for creating and deleting tasks (processes), suspending, and resuming tasks execution. It performs task scheduling using the preemptive priority-based techniques. Each task is assigned a priority when it is created. When more than one task is ready to run, ARTX always selects the highest priority task. Optionally, ARTX schedules tasks of equal priority on a time-sliced basis. When time-slicing is enabled, equal-priority tasks run to the user-specified time-sliced value in round-robin fashion.

4.11 RELIABILITY, ADAPTABILITY, AND MAINTAINABILITY SERVICES

ARTX does not explicitly address these services beyond the standard Ada semantics.

4.12 RESOURCE MANAGEMENT SERVICES

4.12.1 Storage Management

ARTX's approach to dynamic memory allocation is based on the needs of real-time multitasking applications: speed and predictability. Because all variable-block allocation schemes can. under certain conditions, produce unpredictable response times, ARTX allocates and releases memory storage in fixed-size blocks, and a free pool may be subdivided dynamically into partitions to obtain space efficiency. A task can minimize wasted memory by allocating from the partition with block size closest to the actual amount of memory it needs. Therefore, the real-time executive can manage the memory pool without searching overhead and without external fragmentation.

4.13 SCHEDULING SERVICES

ARTX performs task scheduling using the preemptive priority-based techniques. Each task is assigned a priority when it is created. When more than one task is ready to run, ARTX always selects the highest priority task. Optionally, ARTX schedules tasks of equal priority on a time-sliced basis. When time-slicing is enabled, equal-priority tasks run to the user-specified time-sliced value in round-robin fashion.

4.14 SYNCHRONIZATION SERVICES

ARTX provides Dijkstra counting semaphores for mutual exclusion to gain or relinquish exclusive control over a shared resource, such as memory and I/O device, etc. It also uses mailbox and queue for data transfer, synchronization, and mutual exclusion. Event Flags is also used for intertask synchronization.

4.15 SYSTEM INITIALIZATION AND REINITIALIZATION SERVICES

ARTX provides system call for ARTX initialization.

4.16 TIME SERVICES

Real-time clock services are based on the notion of a clock tick; a tick is derived from an interrupt generated by a hardware timer. ARTX maintains a clock counter that accumulates ticks; it increments the counter whenever an interrupt service routine issues a system call. A high-resolution relative time clock like ARTX's is essential for embedded applications.

5.0 SUMMARY

5.1 MACH

Mach has achieved a broad base of interest and support due to its solid technical foundation, UNIX compatibility, and strong backing from DARPA. The UNIX emulation enables Mach to provide an environment in which users may continue to use programs with the UNIX system call interface. This is important for practical reasons because so many UNIX application programs exist.

Mach is a kernel, providing only the basic primitives needed for building distributed and parallel applications. The Mach kernel provides multiple tasks, multiple threads of execution within each task, and synchronous message-based interprocess communication via ports using capabilities to refer to ports.

Mach is an open system based on a lightweight kernel running in each computer with services such as the file system, network service, and process management outside the kernel as independent user-level service tasks. These services replace the system calls found in conventional operating systems such as UNIX. The model provided by Mach is a service model in which objects are managed by servers and clients make request for operations on objects by using remote procedure calls. Remote procedure calling is supported by efficient and flexible interprocess communication facilities in the kernel.

Since Mach makes few traditional operating system decisions within the kernel itself, it is possible to build a completely different operating system environment on top of it. Mach serves as a platform for several interesting distributed system research efforts, many of which are aimed at the NGCR requirements, including real-time computing, reliability/fault tolerance, and security. Our evaluation of Mach (release 2.0) consisted of literature reviews and hands-on. Mach release 2.5 will be available in the fourth quarter of 1989, and release 3.0 is scheduled for spring 1990.

5.1.1 Real-Time

The Advanced Real-time Technology (ART) and Mach groups at CMU are working together on an experiment called RT-Mach (Real-Time Mach) in an attempt to transition some of the ART theoretical work into Mach. They are implementing ART's real-time thread model and scheduler on Mach. This will be for the kernelized version, which is still in development. RT-Mach is being targeted for medium real-time (scheduled events timing 100 ms or greater), so it may be inappropriate for "fast" real-time (robot control or avionics).

5.1.2 Ada

Several Mach mechanisms provide support for Ada (an objective of the Mach project is to support Ada). Mach threads (lightweight processes) are a good match for handling Ada tasking. Mach's flexible virtual memory management and message passing mechanisms (providing dynamic memory allocation, interprocess communication, and exception handling) form a basis for an Ada run-time system.

5.1.3 Distributed

To facilitate building distributed systems, Mach stresses a capability based interprocess communication paradigm, messages containing typed data, and transparent extension of local communication into a network by network message servers. Mach also provides an interface language, which is used to generate client/server interfaces for remote procedure calls, and integration of interprocess communications with virtual memory management for efficient transfer of large messages.

Mach uses the task and threads abstraction, which is a desirable component of a distributed operating system kernel. The usefulness of this abstraction is that by dividing the machine state (thread) from the task, it is now possible to have multiple threads per task, providing support for parallel and distributed processing. The use of threads and the design of system kernels to support them is one of the most significant factors in the success of most of the distributed systems developed to date.

The Mach virtual memory system provides the programmer with a clean interface, which allows user-level programs (external memory managers) to define the exact semantics of virtual memory that can be mapped into any task's virtual address space. External memory management allows Mach to be extended in powerful ways without changing the base Mach kernel. For example, network-consistent shared memory can be implemented by an external memory manager.

A simpler system kernel providing lightweight processes and interprocess communication is a better basis (than UNIX kernel) for building distributed system software. The Mach system software supports this cooperation with appropriate facilities for concurrent processes, interprocess communication, and synchronization.

5.1.4 Security

The Trusted Mach project is to build a version of Mach that meets the B3 level of protection. The design separation of the kernel and services make modification of the operating system more straightforward and easier to verify as being a trusted system. However, the development of a trusted version is tied to the completion of Mach kernelization.

5.1.5 Fault Tolerance

Mach provides the most basic blocks for reliable distributed applications - communication facilities and the Matchmaker interface. Other research projects, such as Camelot, provide functions that support both blocking and nonblocking commit protocols, nested transactions, and a scheme for supporting recoverable objects that are accessed in virtual memory.

5.1.6 Heterogeneity

Mach has achieved high portability. It has been ported to a variety of architectures including Suns, Vax, Encore, Sequent, and BBN Butterfly (multiprocessors). It typically takes less than 3 man-months to port Mach to a new hardware base. Hardware independence was stressed in the design and implementation, especially in the virtual memory management.

5.1.7 Qualification as a Standard

In initiating the Mach project, DARPA aimed to capitalize on the de facto standard status of UNIX. Mach's UNIX compatibility is fundamental to its success and popularity. The Mach project is meant to rebuild the core of UNIX while retaining its external interfaces.

Mach is open and has been widely distributed to corporations, universities, and the government. But, Mach has not yet achieved independence from UNIX. Although the kernelized version of Mach has been planned for some time, it has not yet been implemented and delivered. As operating systems go, Mach is fairly new, and few people understand its potential.

Mach addresses many of the NGCR high-level requirements, but at the kernel level (as opposed to the application interface). As a kernel-level interface (for its basic abstractions: tasks and threads, messages and ports, and virtual memory), Mach would be a good candidate.

5.2 ARTX

Ready Systems' Ada Real-Time Executive (ARTX) is designed to implement the critical "kernel" services of an Ada multitasking real-time Run-Time System for embedded microprocessor applications. ARTX schedules the processor and allocates CPU time among a number of concurrent tasks; it also allocates blocks of available memory to these tasks and implements intertask communication and synchronization. Furthermore, it supports a full range of Ada semantic operations, including the complete Ada tasking model.

5.2.1 Real-Time

ARTX's real-time capabilities include the following: deterministic algorithms with fixed, specified timing for task rescheduling, rendezvous calls and accepts, memory allocation, interrupt latency, and interrupts-off time; a fully preemptive scheduler so the highest priority task in the system will always be executing; and it allows task priority to be changed at run-time.

ARTX provides additional communication and synchronization primitives besides the standard Ada rendezvous. These primitives are accessed from the applications code via a packaged interface and include mailboxes, queues, semaphores, and event flags. It also supports two alternatives for servicing interrupts, using Ada rendezvous entries that are flexible or using Ada procedure calls that are fast but not as flexible.

5.2.2 Ada

ARTX supports the complete Ada tasking operations (e.g., rendezvous, task creation and termination, task completion and abort, select, and delay). It also supports full range Ada semantic operations (e.g., exception handling, input/output, and dynamic memory allocation).

5.2.3 Distributed

ARTX can be configured to provide a multiprocessor networked run-time environment by using its two companion components, RTAda-MP and RTAda-Net. RTAda-MP is for shared memory multiprocessing; and RTAda-Net is for multiprocessor communication over local area networks.

RTAda-MP supports a multiprocessor system organized as a "cluster." A cluster consists of two or more processors with local memory global or shared memory; a system bus that connects the processors; ARTX and RTAda-MP on each processor; and Ada application software. It provides Remote Procedure Calls (RPC) that allow interprocessor procedure calls synchronously or asynchronously.

RTAda-Net allows the ARTX-based system to network with other systems that support TCP/IP and sockets (i.e., Sun/UNIX). For example, real-time applications in an RTAda-Net environment can use ARTX to gather, control, and allocate real-time processes, while a more familiar interface (such as VAX/VMS with TCP/IP, or Unix BSD) can be used for data storage, or as a system monitor in a non-real-time environment.

5.2.4 Heterogeneity

ARTX is targeted to Motorola 68000 family of microprocessors and the Intel 386 microprocessor.

5.2.5 Qualification as a Standard

While ARTX is proprietary, it is also upwardly compatible with Ready Systems' industry standard kernel VRTX, so that application tasks written in other languages (C, Fortran, and assembly language) can be easily integrated into the system without any changes.

ARTX is widely used and includes a sufficient set of NGCR critical capabilities (i.e., real-time, Ada, distributed) to warrant its inclusion as a candidate.

6.0 BIBLIOGRAPHY

Accetta, M. et al. 1986. *Mach: A New Kernel Foundation for UNIX Development,* Computer Science Department, Carnegie Mellon University, Draft Paper.

Baron, R. V. 1988. *MACH Kernel Interface Manual,* Computer Science Department, Carnegie Mellon University, Draft Paper.

C Threads. 1987. Computer Science Department, Carnegie Mellon University, Draft Paper.

Draves, R. R., M. B. Jones, and M. R. Thompson. 1988. *MIG - The MACH Interface Generator,* Computer Science Department, Carnegie Mellon University, Draft Paper.

Gordon, K. D., and C. J. Linn. 1989. *Strategic Defense System Distributed Operating System R&D: Review and Recommendations,* Institute for Defense Analyses (IDA) Paper P-2142.

Jensen, E. D., C. D. Locke, and H. Tokuda. 1985. "A Time-Driven Scheduling Model for Real-Time Operating Systems," *Proceedings of IEEE Real-Time Systems Symposium,* 112-122.

"Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems" 1986. *Proceedings of the 1st Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA).*

MACH Networking Group. 1988. *Network Server Design,* Computer Science Department, Carnegie Mellon University.

Next Generation Computer Resources Operating Systems Standards Working Group. 1989. *Reference Model,* Version 1.02.

Rashid, R. F. 1987. *From RIG to Accent to Mach: The Evolution of a Network Operating System,* Computer Science Department, Carnegie Mellon University.

Rashid, R. et al. 1987. "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures." *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems.*

Ready System's ARTX Reference Manuals.

Sansom, R. D., D. P. Julin, and R. F. Rashid. 1986. "Extending a Capability Based System into a Network Environment," Technical Report CMU-CS-86-115, Computer Science Department, Carnegie Mellon University.

Spector, A. Z., J. J. Bloch, D. S. Daniels, R. P. Draves, D. Duchamp, J. L. Eppinger, S. G. Menees, and D. S. Thompson. 1986. *The Camelot Project,* Computer Science Department, Carnegie Mellon University.

Spector, A. Z., D. Thompson, R. F. Pausch, J. L. Eppinger, D. Duchamp, R. Draves, D. S. Daniels, and J. J. Bloch. 1987. *Camelot: A Distributed Transaction Facility for Mach and the Internet - An Interim Report,* Computer Science Department, Carnegie Mellon University.

Spector, A. Z., and K. R. Swedlow, editors. 1988. *Guide to the Camelot Distributed Transaction Facility: Release 1,* Computer Science Department, Mach/Camelot, Carnegie Mellon University, Draft.

Tevanian, A. Jr. 1987. "Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach", Ph.D. Thesis, Technical Report CMU-CS-88-106, Computer Science Department, Carnegie Mellon University.

Trusted Information Systems, Inc. 1988. *Trusted Mach Presentation,* Ellicott City, Maryland.

Yee, B. S., J. D. Tygar, and A. Z. Spector. 1988. "Strongbox: A Self-Securing Protection System for Distributed Programs", Technical Report CMU-CS-87-184, Computer Science Department, Carnegie Mellon University.

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE November 1989	3. REPORT TYPE AND DATES COVERED FY 89

4. TITLE AND SUBTITLE DISTRIBUTED OPERATING AND RUN-TIME SYSTEMS: MACH AND ARTX	5. FUNDING NUMBERS 0602234N RS 34676 DN 306243
6. AUTHOR(S) D. Brouhard R. Liu M. Gadbois	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Ocean Systems Center San Diego, CA 92152-5000	8. PERFORMING ORGANIZATION REPORT NUMBER NOSC TD 1751
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Naval Ocean Systems Center San Diego, CA 92152-5000	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report addresses research and technical issues in the area of distributed operating run-time systems. Candidate systems, Mach and ARTX are evaluated with respect to their current level of maturity and how well they address the Next Generation Computer Resources (NGCR) requirements for an operating system. The report also describes the NGCR Operating Systems Standards Working Group's abstract model used as a framework for evaluating the candidate systems.

14. SUBJECT TERMS Ready System's Real-Time Ada Executive (ARTX)	15. NUMBER OF PAGES 35
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT SAME AS REPORT